Programare II
Programare Orientată Obiect

Curs 9

Curs anterior

- Moştenire
- Funcţii pur virtuale
- Clase abstracte
- Moştenire multiplă
- Clase de bază virtuale
- RTTI

Curs curent

- Tipuri de date abstracte
- Tipuri de date generice
- Funcţii şablon
- Clase şablon

Tipuri de date

- Realizarea de tipuri de date definite de utilizator care se comportă ca şi tipurile implicite (build-in)
  - De ce proprietăţi avem nevoie;
  - Implementarea unui set de operaţii pentru ele
- Tipuri generice de date
  - Parametrizare astfel încât să funcţioneze cu o mulţime de date şi structuri de date potrivite

Tipuri de date abstracte

Problema	Imlementare
• Creați o clasă care să permită lucrul cu o stivă de întregi	?

Tipuri de date abstracte

Problema

- Creați o clasă care să permită lucrul cu o stivă de întregi

Imlementare

```
class Stiva{
   int *tab;
   int dim, index;

  public:
   Stiva( int d ):index(0), dim(d) { tab = new int[dim]; }
   bool isGoala() {  return index == 0;  }
   bool isPlina() {   return index == dim; }
   void push( int x) {
     if (isPlina()) throw OutOfBounds();
     tab[index++] = x;
   }
   int pop () {
     if (isGoala()) throw OutOfBounds();
       return tab[--index];
     }
  class OutOfBounds{};
};
```

Tipuri de date abstracte

Problema

- Creați o clasă care să permită lucrul cu o stivă de întregi

Dacă vrem o stivă pentru numere reale?

Imlementare

```
class Stiva{
   int *tab;
   int dim, index;

  public:
   Stiva( int d ):index(0), dim(d) { tab = new int[dim]; }
   bool isGoala() {  return index == 0;  }
   bool isPlina() {   return index == dim; }
   void push( int x) {
     if (isPlina()) throw OutOfBounds();
     tab[index++] = x;
   }
   int pop () {
     if (isGoala()) throw OutOfBounds();
       return tab[--index];
     }
  class OutOfBounds{};
};
```

Tipuri de date abstracte

Problema

- Creați o clasă care să permită lucrul cu o stivă de numere reale

Dacă vrem o stivă pentru numere complexe?

Imlementare

```
class StivaDouble{
   double *tab;
   int dim, index;

 public:
   Stiva( int d ):index(0), dim(d) { tab = new double[dim]; }
   bool isGoala() {  return index == 0;  }
   bool isPlina() {   return index == dim; }
   void push( double x) {
     if (isPlina()) throw OutOfBounds();
     tab[index++] = x;
   }
   double pop () {
     if (isGoala()) throw OutOfBounds();
       return tab[--index];
     }
  class OutOfBounds{};
};
```

Tipuri de date abstracte

Cum rezolvăm problema?

Stivă de numere întregi

```
class Stiva{
   int *tab;
   int dim, index;

  public:
   Stiva( int d ):index(0), dim(d) { tab = new int
     [dim]; }
   bool isGoala() {  return index == 0;  }
   bool isPlina() {   return index == dim; }
   void push( int x) {
    if (isPlina()) throw OutOfBounds();
    tab[index++] = x;
   }
   int pop () {
    if (isGoala()) throw OutOfBounds();
      return tab[--index];
    }
  class OutOfBounds{};
};
```

Stivă de numere reale

```
class StivaDouble{
   double *tab;
   int dim, index;

  public:
   Stiva( int d ):index(0), dim(d) { tab = new
     double[dim]; }
   bool isGoala() {  return index == 0;  }
   bool isPlina() {   return index == dim; }
   void push( double x) {
    if (isPlina()) throw OutOfBounds();
    tab[index++] = x;
   }
   double pop () {
    if (isGoala()) throw OutOfBounds();
      return tab[--index];
    }
  class OutOfBounds{};
};
```

Prin ce diferă cele două implementări?

Tipuri de date generice

```
class Stiva<typename T>{
    T *tab;
    int dim, index;
  public:
    Stiva( int d ):index(0), dim(d) {
       tab = new T[ dim];
    }
    bool isGoala() { return index == 0; }
    bool isPlina() { return index == dim; }
    void push( T x) {
      if (isPlina()) throw OutOfBounds();
      tab[index++] = x;
    }
   T pop () {
      if (isGoala()) throw OutOfBounds();
      return tab[--index];
   }
   class OutOfBounds();
};
```

```
int main () {
   Stiva <int> s(4);

   s.push(95);
   s.push(7);
   cout << s.pop();

   Stiva <double> ss(4);

   ss.push(9.5);
   ss.push(7.3);
   cout << ss.pop();
}
```

Tipuri de date generice

- Exprimă algoritmi independenţi de detaliile de reprezentare
- Programare generică
  - se decide algoritmul care se vrea să funcţioneze pentru o varietate de tipuri şi structuri de date
- Definiţie:
  - Un template (şablon, tip de date parametrizat) reprezintă o familie de tipuri sau funcţii, parametrizate cu un tip generic
- Avantaje
  - Reutilizarea codului
  - Permite implementarea de biblioteci cu scopuri generale

Template-uri

- Sintaxă
  - template <listaDeParametri>declaratie;
  - unde
    - listaDeparametrii
      - O listă de parametrii ai template-ului separată prin virgulă
        - Parametrii de tip (class T);
          - T poate fi iniţializat cu un tip de bază (int, char, float),
          - un tip de dată definit de utilizator (MyClass, complex, …),
          - un tip de pointer (void *, …),
          - un tip referinţă (int&, MyClass &, …)
          - o instanţă a unui template
        - Parametrii non-tip (int i);
          - parametri non-tip pot fi instanţiaţi doar cu valori constante şi sunt constanţi în definirea/declararea clasei

Template-uri

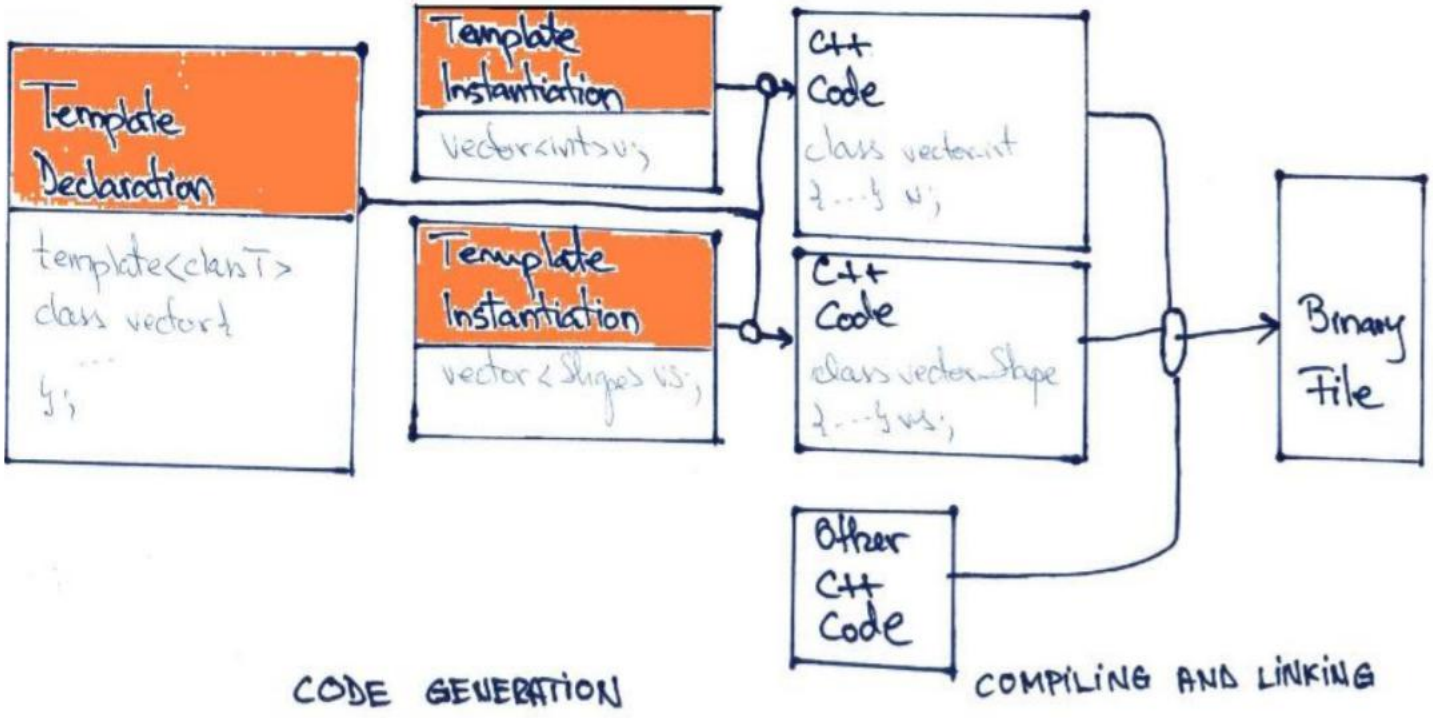
- Template-urile pot fi
  - Clase
  - Funcții

Templeturi-URI

- INSTANŢIEREA
  - Procesul generării unei definiţii de clase dintr-o clasă template
  - Sintaxă
    - DefinireaClaseiTemplate < argumente >declaraţii;
  - Exemple
    - vector <int> d1;
    - vector <double> d2;
    - Buffer <char, 10> d3;
    - MyClass <int, Employee, 10, 0.5>x;
    - MyClass<Employee&, Manager*, 20-1, 103/7> y;

Generarea codului

- Compilatorul C++ generează cod doar pentru clasele/funcţiile utilizate.



Generarea codului

- Compilatorul va genera declaraţii de clase doar pentru template-urile instanţiate
- Funcţiile ‘obişnuite’ sunt generate doar pentru funcţiile membre ale template-ului utilizare
- Dacă template-ul nu este instanţiat nu se generează cod

Generarea codului

Exemplu

- Exemplu

```
int main (int, char*[]) {
    vector  <int>v0, v1;
    v.add(1) ;
    v.add(100);
    cout << v.get (0);
    v1 = v0;
    vector <float>fv;
    return 0;
}
```

Ce cod se va genera?

- ?

Generarea codului

Exemplu

- Exemplu

```
int main (int, char*[]) {
    vector  <int>v0, v1;
    v.add(1) ;
    v.add(100);
    cout << v.get (0);
    v1 = v0;
    vector <float>fv;
    return 0;
}
```

Ce cod se va genera?

- Pentru clasa vector <int>
  - declaraţia clasei (include funcţiile inline)
  - operatorul de atribuire
  - funcţia add
  - funcţia get
- Pentru clasa vector <float>
  - declaraţia clasei (include funcţiile inline)

Verificarea tipului

- Erori în definirea unui template
  - Care pot fi determinate la compilare, exemplu punct şi virgulă sau cuvinte cheie scrise greşit
  - Care pot fi identificate la instanţierea template-ului (exemplu de mai jos)
  - Care pot fi identificate la execuţie

- Punct de instanţiere
  - Prima instanţiere a unui template, utilă pentru detectarea şi rezolvarea erorilor din template
  - Pentru depanare se util de folosit tipurile cele mai frecvente

Verificarea tipului

Argumentele pasate la template-uri trebuie să aibă operaţiile cerute de template

```
class X { };

template void vector::add(T x) {
    // add e to the vector v
  std::cout << "Added element "<< x;
}
```

```
void f1() {
   vector vi; // instantiere
   vi.add(100); // => OK!
}

void f2() {
    vector vX; // instantiere
     vX.add(X(7)); // => eroare!
   //De ce este eroare?
}
```

Funcții template

```
int min( int x, int y) { return x<y?x:y; }

float min( float x, float y) { return x<y?x:y; }

complex& min(complex& x, complex& y) {
        return x<y?x:y;
}

void f() {
   complex c1(1,1), c2(2,3);
   min(0,1);
   min(6.5, 3);
   min(c1, c2);
}
```

Cum putem grupa cele 3 funcții?

Funcții template

Fară folosirea template-uri

```
int min( int x, int y) { return x<y?x:y; }

float min( float x, float y) {
      return x<y?x:y;
}

complex& min(complex& x, complex& y)
    {
      return x<y?x:y;
}

void f() {
  complex c1(1,1), c2(2,3);
  min(0,1);
  min(6.5, 3);
  min(c1, c2);
}
```

Folosind template-uri

```
template <typename T> min( T x, T y) {
  return x<y?x:y;
}

void f() {
  complex c1(1,1), c2(2,3);
  min(0,1);
  min<float>(6.5, 3);
  min(c1, c2); // min<complex>(c1,c2);
}
```

Funcții template

```
template <class T> min( T x, T y);

template <class T> min( T x, T y) {
  return x<y?x:y;
}

void f() {
  complex c1(1,1), c2(2,3);
  min(0,1);
  min<float>(6.5, 3);
  min(c1, c2);
  min<complex>(c1,c2);
}
```

- Sintaxă
  - template < listaTipuriParametrii > numeFunctie(lista de parametrii);

Funcţii template

- INSTANŢIERE ŞI AMBIGUITĂŢI
  - Apelarea funcţiilor template
    - min <int>(0,1)
    - min <complex>(complex(2,3), complex(3,4))
  - Dacă argumentele corespund tipurilor de date ale templateului (ex. pentru funcţia min ambele argumente sunt de acelaşi tip) compilatorul va instanţia automat funcţia fără a mai fi nevoie ca utilizatorul să specifice explicit tipurile parametrilor
  - Exemplu
    - min (0,1); //OK
    - min (2.5, 4); //abigu; este nevoie de un apel explicit min (2.5, 4)
    - min (2.5, 4); //OK

Funcţii template

- INSTANŢIERE ŞI AMBIGUITĂŢ
  - Rezolvarea ambiguităţilor
    - Apelarea explicită
    - suprîncărcarea / specializarea prin adăugarea unor noi funcţii care să se potrivească cu apelul (ex. double min(double, int))

Funcţii template

- SUPRAÎNCĂRCARE. PARAMETRI MULTIPLI
  - Supraîncărcarea funcţiilor template
    - Funcţia supraîncărcată ascunde funcţia template
  - Template-urile pot accepta mai multe tipuri generice
    - Sintaxă
      - template <class tip1, class tip2, ..., class tipN> numeFunctie (listaDeParametrii);
    - Un număr mare de parametrii poate produce confuzii
    - Tipul de return dacă este generic trebuie să se regăsească în lista de tipuri
    - Exemplu
      - template <class T,1 class T2> T1 add(T1 a, T2 b);

Clase generice

- Sintaxă
  template <class T [, class T1[, ...]]>
  class nume_clasa { …}

- Declarare metode .h
  template < class T > class MyClass {
  // Folosirea lui T ca un tip obisnuit
  bool test(T item);
  };

- Definire metode.h
  template < class T > bool MyClass::test(T item) {
  // Folosirea lui T ca un tip obisnuit
  }

Clase generice

- Moştenire
  - Moştenirea funcţionează la fel ca în cazul claselor ‘obişnuite’
- Parametrii default
  - O valoare default poate fi specificată din în definiţia template-ului
  - Exemplu
    - template <class T1, class T2 = int> class MyClass {… }

CURS VIITOR

- Stanard Template Library
  - Structuri de date comune
    - List, map ,…
  - Iteratori
  - Algoritmi